COO-3077-28

Courant Institute of
Mathematical Sciences

AEC Computing and Applied Mathematics Center

# A Simple Lambda-Calculus Model of Programming Languages

S. Kamal Abdali

AEC Research and Development Report

Mathematics and Computers
July 1973

New York University

AEC Computing and Applied Mathematics Center
Courant Institute of Mathematical Sciences
New York University

Mathematics and Computers COO-3077-28

# A SIMPLE LAMBDA-CALCULUS MODEL OF PROGRAMMING LANGUAGES

S. Kamal Abdali

July 1973

Contract No. AT(11-1)-3077

## ABSTRACT

We present a simple correspondence between a large subset of ALGOL 60 language and lambda-calculus. With the aid of this correspondence, a program can be translated into a single lambda-expression. In general, the representation of a program is specified by means of a system of simultaneous conversion relations among the representations of the program constituents. High-level programming language features are treated directly, not in terms of the representations of machine-level operations. The model includes input-output in such a way that when the representation of a (convergent) program is applied to the input item representations, the resulting combination reduces to a tuple of the representations of the output items. This model does not introduce any imperative notions into the calculus; the descriptive programming constructs, such as expressions, and the imperative ones, such as assignments and jumps, are translated alike into pure lambda-expressions. The applicability of the model to the problems of proving program equivalence and correctness is illustrated by means of simple examples.

CONTENTS

# 1. INTRODUCTION

If one wishes to study properties of programs, then one should either develop rules of deduction and inference that apply directly to programming language constructs (e.g., Hoare [1]), or represent programs by the objects of some mathematical system [e.g., 2,3,7-10] and work with these representations. As a step in the second direction, this paper describes a way of representing programs by lambda-expressions [4-6].

Since a number of lambda-calculus (or, related, combinatory logic) models of programming languages have already appeared in the literature [among them, 7-10], the proposal of yet another such model may require justification. So we will first indicate some distinguishing features of our model vis-à-vis others.

1. Our model does not introduce any imperative or otherwise foreign notions to lambda-calculus. This is in contrast to Landin [7], in which imperative features of programming languages are accounted for by ad hoc extensions of the calculus. We find that the calculus, in its purity, suffices as a model of programming languages. By not making any additions to the lambda-calculus, we have the guarantee that all its properties, in particular, the consistency and the Church-Rosser property [6], are valid in our model. For example, even when a program requires a fixed order of execution, the lambda-expression obtained by evaluating the program representation in any order, whatsoever, represents the program result correctly.

2. Programs are <u>translated</u> into lambda-expressions, not <u>interpreted</u> by a lambda-calculus interpreter (Reynolds [10]). Thus, programming semantics is completely reduced to lambda-calculus semantics, but without commitment to any particular view of the latter. Also, all lambda-expression transformations are applicable to programs.

3. Assignments are modelled by the substitution operation of lambda-calculus. Consequently, the notions of memory, addresses, and fetch and store operations do not enter our model in an explicit manner (Stratchey [8], Reynolds [10]).

4. High-level programming language constructs are represented by lambda-expressions directly, not in terms of the representations of the machine-level operations (Orgass-Fitch [9]) of the compiled code.

5. This model spans the full ALGOL 60 language. It is also applicable to a number of other advanced programming features, such as collateral (parallel) statements, the use of labels and procedures as assignable values, coroutines, etc.

6. As a matter of opinion, it seems that our representations are much simpler and clearer than the ones given in other models. Also they seem to have obvious intuitive justifications.

The model will be described informally, and only for a representative set of programming language constructs. But enough motivating details and illustrations will be provided to, hopefully, convey the method, and suggest its extension to other programming features. A more complete treatment can be found in [11].

## 2. DESCRIPTION OF THE MODEL -- BASIC FEATURES

### 2.1 Preliminaries

To fix our terminology and notation for lambda-calculus (LC), we include the following definitions.

We assume given a set of entities, called variables. A lambda-expression (LE) is either a variable, or the application $(e_1 e_2)$ of an LE $e_1$ to an LE $e_2$, or the abstraction $(\lambda x{:}e)$ of an LE $e$ with respect to a variable $x$. In writing LE's, parentheses may be omitted under the convention that applications associate to the left and abstractions to the right, with the former taking precedence over the latter. For instance,

$$(\lambda x{:}(\lambda y{:}(\lambda z{:}(((x\ z)(y\ z))u))))$$

may be abbreviated to

$$\lambda x{:}\ \lambda y{:}\ \lambda z{:}\ xz(yz)u\ .$$

As an additional convention, the above may be further abbreviated to

$$\lambda xyz{:}\ xz(yz)u\ .$$

Identity of LE's is indicated by the symbol '$\equiv$', which is also used as a definition symbol.

In the LE $(\lambda x{:}e)$, the first occurrence of $x$ is a binding occurrence, and $e$ is the range of that occurrence. An occurrence of a variable in an LE is bound if it is either binding or in the range of any binding occurrence of the same variable; otherwise, the occurrence is free. If $e, f_1, \ldots, f_n$ are LE's and $x_1, \ldots, x_n$ variables, then

$$\mathrm{sub}[f_1, x_1; \ldots; f_n, x_n; e]$$

denotes the result of simultaneously substituting $f_i$ for all free occurrences of $x_i$ $(1 \leq i \leq n)$ in $e$.

The basic LC rules of transformation, called contractions, are these:

($\alpha$) $\lambda x{:}e \rightarrow_{\alpha} \lambda y{:}\ sub[y,x;e]$,

if y has no free occurrences in e, and no free occurrences of x become bound occurrences of y by the substitution.

($\beta$) $(\lambda x{:}e)f \rightarrow_{\beta} sub[f,x;e]$,

if no variable with free occurrences in f has bound occurrences in e.

($\eta$) $\lambda x{:}\ ex \rightarrow_{\eta} e$,

if x has no free occurrences in e.

The converse of $\beta$- (or $\eta$-) contraction is called $\beta$- or $\eta$-expansion. Contractions and expansions may be applied to the whole or a part (which must itself be an LE) of an LE. Reduction ($\rightarrow$) consists of a (possibly empty) sequence of contractions. As an example of reduction, it can be shown that

$$(\lambda x_1 \ldots x_n{:}\ e)\ f_1 \ldots f_n \rightarrow sub[f_1,x_1;\ldots;f_n,x_n;e]\ ,$$

provided that no variable with free occurrences in f's has bound occurrences in e. Conversion (=) consists of a (possibly empty) sequence of contractions and expansions. If it is the case that e = f, then there exists an LE g such that $e \rightarrow g$ and $f \rightarrow g$ (Church-Rosser Theorem [6]). An LE is irreducible if no $\beta$- or $\eta$-contraction is applicable to it, even after any intermediate $\alpha$-contractions. A normal form of an LE e is an irreducible LE g, if one exists, such that e = f. If e possesses a normal form f, then f is unique up to the applications of $\alpha$-contractions, and, moreover, $e \rightarrow f$; f can be obtained from e by using, among other possibilities, the standard-order reduction algorithm, in which one always chooses the leftmost $\beta$- or $\eta$-contraction at each step in reduction.

A natural interpretation of LE's is as functions: the LE (f e) may be regarded as the functional expression f(e), and $(\lambda x{:}e)$ as the function f defined by f(x) = e in the customary functional notation. With this interpretation, the LE $(e_1e_2e_3\ldots e_n)$ may be

viewed variously as $e_1(e_2,\ldots,e_n)$, or $[e_1(e_2)](e_3,\ldots,e_n)$, where $e_1(e_2)$ is understood to yield a function as its value, etc. The functional interpretation of LE's is the intuitive basis of our model.

Following are some definitions and abbreviations that will be used later:

$$\langle a_1,\ldots,a_n\rangle \equiv \lambda x: xa_1\ldots a_n \ , \quad n \geq 0$$

$$I \equiv \lambda x:x$$

$$\Omega \equiv (\lambda xy:xx)(\lambda xy:xx)$$

$$Y \equiv \lambda x: (\lambda y: x(yy))\ (\lambda y: x(yy))$$

$$\underline{a,b} \equiv \lambda x: axb$$

$$\underline{elem}_{i,n} \equiv \lambda x_1 \ldots x_n: x_i \ , \quad 1 \leq i \leq n$$

$$\underline{replace}_{i,n} \equiv \lambda y x_1 \ldots x_n: \langle x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n\rangle, \quad 1 \leq i \leq n.$$

Detailed information about LC may be found in [4-6].

We use the ALGOL 60 notation [12], whenever possible, to express programming language constructs. By the environment of a point in a program, we mean a list of all the variables whose scopes include that point; in this list, the variables are arranged in their order of declaration within individual blocks, with the blocks taken in the innermost to outermost order. In general, the LC representation of a construct may depend on its own environment (i.e., of the point at which it occurs), as well as the environments of its components. We denote the LC representation of a construct X appearing in the environment E by $\{X\}_E$, or simply $\{X\}$, when the environment is understood, or is irrelevant to the representation, such as when X is a constant.

## 2.2 Programs as Lambda-Expressions

Before we can choose the LC representations of various programming language constructs, we must decide exactly how we are to represent complete programs. We will take the view that it is the external input-output behavior that most appropriately characterizes a program. Consider, for example, the program

```
begin integer a,b,c;
   read a; read c; b := a+c; write b; b := b-2xc; write b
end
```

in which read and write are assumed to be the standard statements for performing input-output operations. As far as only the external input-output is concerned, the above program behaves as a two-argument function which produces as value two quantities, the sum and the difference of the arguments; in LC notation, this function can be expressed by

$$\lambda xy: \langle x+y,\ x-y \rangle \quad . \tag{1}$$

(Note that x+y and x-y are not strictly LE's, but they can be easily translated to be such.) Accordingly, we would like to set up the model in such a manner that the representation of the above program may turn out to be the LE (1), or, as seems more likely, an LE which can be reduced to (1). So constructed, the model would enable us, in essence, to abstract out of a program code the function from the input space to the output space that the program computes.

More generally, if P is a program, $i_1,\ldots,i_p$ its inputs, and $o_1,\ldots,o_q$ its outputs, then our representations are intended to satisfy the reduction relation

$$\{P\}\{i_1\}\ldots\ \{i_p\} \rightarrow \langle\{o_1\},\ldots,\{o_q\}\rangle \quad . \tag{1a}$$

While the main goal of the present model is to obtain purely functional descriptions of programs, it will become evident later that the representations in this model are also capable of simulating the execution-time behavior of programs. Specifically,

when a slightly modified version of the standard-order reduction algorithm is employed to carry out (1a), it, in effect, furnishes the step-by-step trace of executing the program P with the input data items $i_1,\ldots,i_p$.

## 2.3 Variables

Program variables are represented by LC variables, possibly with the same symbolic denotation. However, all program variables with non-disjoint scopes must have mutually distinct representations; thus, if the same identifier is used for two such variables, the corresponding LC representations must be distinguished from each other, say, by superscripting the identifier with the respective block level numbers.

A single LC variable is used to represent an array also; however, the treatment of arrays is deferred to a later section, and until that discussion we assume all variables to be simple.

## 2.4 Constants, Operations, Relations

We do not concern ourselves here with the purely arithmetical aspects of programming language; so we take for granted the LC representations of numbers, logical values, arithmetic and logical operations, and relations. (The representation method may be found in [4,11].) We assume that these representations satisfy the reduction relations of the form

$$\{*\}\ \{a\} \rightarrow \{*\ a\}\ ,$$

$$\{\circ\}\ \{a\}\ \{b\} \rightarrow \{a \circ b\}\ ,$$

where $*$ and $\circ$ are unary and binary operators, respectively, and $a$ and $b$ are the quantities of proper types. (Note: $\{X\}$ denotes the LC representation of X, ref. end of Sec. 2.1.) The representations of logical values are assumed to satisfy

$$\{\underline{\text{true}}\}\ \{a\}\ \{b\} \rightarrow \{a\}$$
$$\{\underline{\text{false}}\}\ \{a\}\ \{b\} \rightarrow \{b\}$$

for all a and b.

## 2.5 Expressions

Let us limit ourselves for the present to the expressions that do not contain function designators. (This restriction will be lifted later when we discuss procedures.) In view of the previous section, the representation of such expressions is essentially a matter of rearranging the individual operator and operand representations into a parenthesized prefix form. For example,

$$a + \underline{\text{if}}\ b \neq 0\ \underline{\text{then}}\ c/b\ \underline{\text{else}}\ 15 \qquad (2)$$

is represented by

$$\{+\}\{a\}(\{\neq\}\{b\}\{0\}(\{/\}\{c\}\{b\})\{15\}) \ . \qquad (3)$$

For convenience in reading, we utilize Landin's idea of "syntactic sugaring" [7]: we designate (3) by simply underlining (2).

## 2.6 Assignments

Before discussing any particular type of statements, we will first indicate the general idea behind our LC representations. Consider a given statement S of a program. Let $(v_1,\ldots,v_n)$ be the environment of S, and denote by F the section of the program following S and extending all the way to the program end. (F will be referred to as the program remainder of S.) The two parts of the program, one consisting of F alone, and the other composed of S and F together, may be interpreted as two functions $\phi$ and $\phi'$, respectively, of the arguments $v_1,\ldots,v_n$. With this interpretation in mind, the effect of the statement S is to transform $\phi$ into $\phi'$. As the representation of S, therefore, we take precisely the function (to be accurate, the functional operator) $\sigma$ given by

$$[\sigma(\phi)](v_1,\ldots,v_n) = \phi'(v_1,\ldots,v_n) \ ,$$

which accomplishes the above transformation.

A key step in representing a programming operations is to find a suitable $\phi'$ for it, based on the intuitive understanding of the effect of the operation. Purely for the sake of motivation, we will include details of this step in describing the first few of our representations.

Let us now turn to the assignment statement $v_i := e$ in the environment $(v_1,\ldots,v_n)$. The $\phi'$ in this case differs from $\phi$ in having the argument $v_i$ set to e. Thus, in effect, the above statement behaves as the function $\sigma$ such that

$$[\sigma(\phi)](v_1,\ldots,v_n) = \phi'(v_1,\ldots,v_n) = \phi(v_1,\ldots,v_{i-1},e,v_{i+1},\ldots,v_n) .$$

Accordingly, we adopt the following LC representation of assignment statements.

$$\{v_i := e\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n : \phi v_1 \ldots v_{i-1}\{e\}v_{i+1} \ldots v_n . \quad (4)$$

(An assumption underlying (4) is that any needed type-conversion has been incorporated within e itself.)

If the expression e contains only the variables $v_1,\ldots,v_m$ for some $m < n$, then the variables $v_{m+1},\ldots,v_n$ can be dropped from the above LE (by $\eta$-contraction), reducing it to

$$\lambda\phi v_1 \ldots v_m : \phi v_1 \ldots v_{i-1}\{e\}v_{i+1} \ldots v_m .$$

Note that the multiple assignment of ALGOL 60 and the collateral (parallel) assignment of ALGOL 68 pose no special problem. As an illustration of the latter, the statement (x := y, z:=x) occurring in the environment (x,y,z) has the LC translation $\lambda\phi xyz:\phi yyx$.

From (4) it is clearly seen that the LE representing an assignment statement does not contain any free variables, as $v_1,\ldots,v_n$ are the only variables that can legitimately occur in e. This property of containing no free variables will be seen to be enjoyed by the LC representation of all types of statements, and will be used to simplify the representations.

## 2.7 Compound Statements

Consider the compound statement $S \equiv \underline{\text{begin}}\ S_1;S_2\ \underline{\text{end}}$ appearing in the environment $(v_1,\ldots,v_n)$. Let $F$ be the segment of the program that follows $S$. The execution of $S;F$ has precisely the same effect as that of $S_1;S_2;F$. We can imagine the program sections $F$, $(S_2;F)$, and $(S_1;S_2;F)$ to be some functions $\phi$, $\phi"$, and $\phi'$, respectively, of the arguments $v_1,\ldots,v_n$, and $S$, $S_1$, $S_2$ to be the functional operators $\sigma$, $\sigma_1$, $\sigma_2$ such that

$$[\sigma_2(\phi\ )](v_1,\ldots,v_n) = \phi"(v_1,\ldots,v_n)\ ,$$

$$[\sigma_1(\phi")](v_1,\ldots,v_n) = \phi'(v_1,\ldots,v_n)\ ,$$

$$[\sigma\ (\phi\ )](v_1,\ldots,v_n) = \phi'(v_1,\ldots,v_n)\ .$$

In LC notation,

$$\phi" \equiv \lambda v_1\ \ldots v_n\colon \sigma_2\phi\ v_1\ldots\ v_n \rightarrow \sigma_2\phi\ ,$$

as the statement representations $\sigma_2$ and $\phi$ do not contain free variables; hence,

$$\sigma \equiv \lambda\phi v_1\ldots\ v_n\colon \phi' v_1\ldots\ v_n \equiv \lambda\phi v_1\ldots\ v_n\colon \sigma_1\phi" v_1\ldots\ v_n$$

$$\rightarrow \lambda\phi v_1\ldots\ v_n\colon \sigma_1(\sigma_2\phi)v_1\ \ldots v_n \rightarrow \lambda\phi\colon \sigma_1(\sigma_2\phi)\ .$$

The generalization to an n-component compound is obvious. So we are led to the following LC representation of compound statements.

$$\{\underline{\text{begin}}\ S_1;S_2;\ldots;S_n\ \underline{\text{end}}\} \equiv \lambda\phi\colon \{S_1\}(\{S_2\}(\ldots(\{S_n\}\phi)\ldots)) \qquad (5)$$

Notice the convenient fact that the individual statement representations in (5) appear from left to right in the same order in which the statements occur in the compound (Cf. Stratchey [8]).

Example: Rules (4) and (5) are illustrated by the following statements and their LC representations. The environment is assumed to be (x,y).

| | Statements | LC Representations |
|---|---|---|
| (a) | begin | |
| | x := 2; | $\lambda\phi xy$: $\phi\underline{2}$ y $\equiv$ A, say |
| | y := x+3; | $\lambda\phi xy$: $\phi$x $\underline{x+3}$ $\equiv$ B |
| | x := y+x | $\lambda\phi xy$: $\phi$ $\underline{x+y}$ y $\equiv$ C |
| | end | $\lambda\phi$: A(B(C $\phi$)) $\equiv$ D |
| (b) | begin | |
| | y := 5: | $\lambda\phi xy$: $\phi$x $\underline{5}$ $\equiv$ E |
| | x := y+2 | $\lambda\phi xy$: $\phi$ $\underline{y+2}$ y $\equiv$ F |
| | end | $\lambda\phi$: E(F $\phi$) $\equiv$ G |

It can be verified that the LC representations of the equivalent program segments (a) and (b), namely, D and G, are indeed mutually convertible LE's; specifically, D $\rightarrow$ $\lambda\phi xy$: $\phi$ $\underline{7}$ $\underline{5}$ $\leftarrow$ G. The normal form $\lambda\phi xy$: $\phi$ $\underline{7}$ $\underline{5}$ of these representations corresponds to the collateral assignment statement (x := 7, y := 5), which may be thought of as the simplest version of the above code.

## 2.8. Blocks.

Next, let us consider a block S whose head declares the variables $u_1,\ldots,u_m$, and initializes these to the values $c_1,\ldots,c_m$, and whose body consists of the statements $S_1,\ldots,S_p$, in that order. The execution of S can be broken down into three operations performed in succession.

1) Extension of the existing environment by the variables $u_1,\ldots,u_m$ (initialized at $c_1,\ldots,c_m$).
2) Execution of the compound begin $S_1;\ldots;S_p$ end .
3) Deletion of the variables $u_1,\ldots,u_m$ from the environment.

Let these three operations be denoted by the functions $\alpha$, $\beta$, and $\gamma$. Let $(v_1,\ldots,v_n)$ be the environment of S. Then with the obvious significance of other symbols, we have

$$[\alpha(\phi)](v_1,\ldots,v_n) = \phi(c_1,\ldots,c_m, v_1,\ldots,v_n)$$

$$[\beta(\phi)](u_1,\ldots,u_m,v_1,\ldots,v_n)$$

$$= [\sigma_1(\sigma_2(\ldots(\sigma_\phi(\phi))\ldots))](u_1,\ldots,u_m,v_1,\ldots,v_n)$$

$$[\gamma(\phi)](u_1,\ldots,u_m,v_1,\ldots,v_n) = \phi(v_1,\ldots,v_n)$$

$$[\sigma(\phi)](v_1,\ldots,v_n) = [\alpha(\beta(\gamma(\phi)))](v_1,\ldots,v_n) .$$

By expressing the above in LC notation, and making use of proper substitutions and simplifications, we obtain

$$\sigma \equiv \lambda\phi v_1\ldots v_n\colon \sigma_1(\sigma_2(\ldots(\sigma_p(\lambda u_1\ldots u_m\colon \phi))\ldots))c_1\ldots c_m\ v_1\ldots v_n .$$

Consequently, we choose the following representation of blocks:

$$\{\underline{\text{begin}}\ \langle\text{type}\rangle u_1:=c_1;\ldots;\langle\text{type}\rangle u_m:=c_m;S_1;S_2;\ldots;S_p\ \underline{\text{end}}\}(v_1,\ldots,v_n)$$

$$\equiv \lambda\phi v_1\ldots v_n\colon \{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(\lambda u_1\ldots u_m\colon\phi))\ldots))$$

$$\{c_1\}_E\ \ldots\{c_m\}_E\ v_1\ \ldots\ v_n\ ,$$

$$\text{where}\quad E \equiv (v_1,\ldots,v_n)\ \text{and}\quad F \equiv (u_1,\ldots,u_m,v_1,\ldots,v_n). \tag{6}$$

In (6), we assume that the expressions $c_i$ include any required type-conversion operations. In the case that the variables are left uninitialized in the block-head, any arbitrary LE can be used in place of $c_i$. One might wish to use for this purpose an LE which would play the role of the everywhere undefined function. This function is modelled, for example, by the LE

$\Omega \equiv (\lambda xy{:}xx)(\lambda xy{:}xx)$, which has the property $\Omega x \rightarrow \Omega$ for all x. It should be noted, however, that $\Omega$ does not possess a normal form (as can be easily verified). As a result, if $\Omega$ is used in place of the missing $c_i$'s in (6), then the presence of any variables that remain undefined throughout the program execution would cause the program representation to behave as if the program contained an infinite loop.

If the variables declared in a block-head are not initialized, then $v_i$ $(1 \leq i \leq n)$ have no free occurrences in the expression at the right-hand of (6), and we have the simpler representation

$$\{\underline{\text{begin}}\ \langle\text{type}\rangle u_1;\ldots;\langle\text{type}\rangle u_m;S_1;S_2;\ldots;S_p\ \underline{\text{end}}\}(v_1,\ldots,v_n)$$

$$\equiv \lambda\phi\colon \{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(\lambda u_1 \ldots u_m{:}\phi))\ldots)) \underbrace{\Omega\ \Omega\ \ldots\Omega}_{m \text{ times}},$$

$$\text{where } F \equiv (u_1,\ldots,u_m,v_1,\ldots,v_n). \qquad (7)$$

## 2.9 Input-Output

We shall assume for simplicity that the program input and output operations are each restricted to a single file. A file of items $a_1,\ldots,a_n$ will be represented by the tuple

$$\langle\{a_1\},\ldots,\{a_n\}\rangle, \quad \text{i.e.,} \quad \lambda x\colon x\{a_1\} \ldots \{a_n\}\ .$$

(The empty file is represented by the null tuple $\langle\ \rangle \equiv \lambda x\colon x \equiv I$.) For given LE's u, v, and w, we will abbreviate the LE $\lambda x\colon uxv$ by $\underline{u,v}$ and the LE $\underline{u,v,w}$ by $\underline{u,v,w}$. It can be easily verified that

$$\underline{\langle x_1,\ldots,x_n\rangle,y} \rightarrow \langle x_1,\ldots,x_n,y\rangle$$

$$\underline{I,x_1,\ldots,x_n} \rightarrow \langle x_1,\ldots,x_n\rangle\ ,$$

so that "," may be regarded as the operation of writing on a file, and the file resulting from writing an item a on a given file b may be represented by $\underline{\{b\},\{a\}}$.

Now let S be a statement appearing in the environment $v_1,\ldots,v_n$ of a program, and let $\sigma$ be the LC representation of S. In our disucssion so far, $\sigma$ has been an LE of the form

$$\lambda\phi v_1 \ldots v_n : \ldots \quad , \tag{*}$$

with the argument $\phi$ standing for the program remainder of S. Accordingly, the execution of S has been modelled by the reduction of the LE

$$\sigma \; \underline{\phi} \; \underline{v}_1 \; \cdots \; \underline{v}_n \; ,$$

in which the underlined symbols denote the values of the corresponding arguments immediately prior to the execution of S. In order to take input-output into account, we will generalize the representations so as to model the above execution by the reduction of the LE

$$\sigma \; \underline{\phi} \; \underline{v_1} \; \ldots \; \underline{v_n} \; \{w\}\{u_1\}\{u_2\} \; \ldots \; \{u_m\} \quad , \tag{8}$$

with w denoting the output file and $u_i$ the i-th of the m items remaining on the input file at the moment of execution. (As soon as an item is read, it is supposed to disappear from the input file.) This arrangement requires that the representations of statements be generally of the form

$$\lambda\phi v_1 \ldots v_n \; o \; i_1 \ldots i_m : \ldots \quad ,$$

where $o, i_1,\ldots,i_m$ are the extra variables denoting the output file and input items. It must be evident, however, that the representations of those statements which do not involve input-output can be simplified back to the form (*) by the use of $\eta$-contractions. Furthermore, in the case of input-output statements, the following choice of LC representations is obvious.

$$\{\underline{\text{read}}\ v_j\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n o i : \phi v_1 \ldots v_{j-1} \, i \, v_{j+1} \ldots v_n o \tag{9}$$

$$\{\underline{\text{write}}\ e\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n o : \phi v_1 \ldots v_n \; \underline{o,\{e\}} \quad , \tag{10}$$

where e is some expression to be output.

## 2.10 Programs

Let the input file initially presented to a given program consist of items $i_1,\ldots,i_p$ , and let $o_1,\ldots,o_q$ constitute the items of the final output file produced by the program. As remarked in Sec. 2.2, we wish to choose a program representation so as to obtain the relation

$$\{\text{program}\}\{i_1\}\ldots\{i_p\} \rightarrow \langle\{o_1\},\ldots,\{o_q\}\rangle \quad . \tag{11}$$

Now the execution of a particular statement of the program is simulated by an LE given by (8) in the previous section. Suppose that as an instance of such a statement we take the entire outermost block of the program. Recalling the significance of symbols used in connection with (8), we obtain the following conditions:

$\sigma \equiv$ {program block},

$n = 0$ as the environment is null,

$\{w\} \equiv I$, as the output file may be considered empty at the start of the program,

$m = p$, and $u_j = i_j$ , $1 \leq j \leq p$ .

Furthermore, in place of $\underline{\phi}$ , the "null" program remainder, we may arbitrarily choose to employ the LE $I \equiv \lambda\phi\colon \phi$ . On substituting these values, the execution of the program is seen to amount to the reduction of the LE

$$\{\text{program block}\}\ I\ I\ \{i_1\}\ \ldots\ \{i_p\} \quad . \tag{12}$$

Next, consider (8) again -- but this time for the case when the entire program has been executed. Now we have:

$\sigma \equiv I$, the null program segment,
$n = 0$, as the environment is null,
$\{w\} = \langle\{o_1\},\ldots,\{o_q\}\rangle$ , representing the final output file,
$m = 0$, assuming the program exhausts the input file,
$\phi \equiv I$ .

Thus, (8) in this case becomes the LE

$$I\ I\ \langle\{o_1\},\ldots,\{o_q\}\rangle$$

which reduces to

$$\langle\{o_1\},\ldots,\{o_q\}\rangle \quad . \tag{13}$$

If our representations work properly, then the LE (12) should reduce to the LE (13). Comparing this reduction relation with (11), we obtain

$$\{\text{program}\} \equiv \{\text{program block}\}\ I\ I \quad . \tag{14}$$

Example. To illustrate the parts of the model introduced so far, we will describe in some detail the representation of the simple program mentioned at the beginning of Section 2.2. The components of the program and their representations are shown side by side below.

| Statements | Representations |
|---|---|
| begin integer a,b,c; | |
| read a; | $\lambda\phi abcoi{:}\phi ibco \equiv A$ , say |
| read c; | $\lambda\phi abcoi{:}\phi abio \equiv B$, |
| b := a+c; | $\lambda\phi abc{:}\phi a\ \underline{a+c}\ c \equiv C$ , |
| write b; | $\lambda\phi abco{:}\phi abc\ \underline{o,b} \equiv D$ , |
| b := b-2xc; | $\lambda\phi abc{:}\phi a\ \underline{b-2xc}\ c \equiv E$ , |
| write b | $\lambda\phi abco{:}\phi abc\ \underline{o,b} \equiv F$ , |
| end | $\lambda\phi{:}\ A(B(C(D(E(F(\lambda abc{:}\phi)))))))\Omega\Omega\Omega \equiv G$ . |

Since G represents the program block, the representation of the whole program is G I I. It can be easily verified that

$$G\ I\ I \rightarrow \lambda xy{:}\langle\underline{x+y},\underline{x-y}\rangle \quad .$$

Thus GII indeed abstracts out the input-output behavior of the program. (Cf. the discussion in Sec. 2.2.) The execution trace of the above program, when run with the data items 5 and 3, is reflected in the following LC reduction sequence.

$$
\begin{aligned}
\mathrm{GII}\ \underline{5}\ \underline{3} &\to A(B(C(D(E(F(\lambda abc{:}\ I))))))\Omega\Omega\Omega I\ \underline{5}\ \underline{3} \\
&\to B(C(D(E(F(\lambda abc{:}\ I)))))\ \underline{5}\ \Omega\ \Omega\ I\ \underline{3} \\
&\to C(D(E(F(\lambda abc{:}\ I))))\ \underline{5}\ \Omega\ \underline{3}\ I \\
&\to D(E(F(\lambda abc{:}\ I)))\ \underline{5}\ \underline{5{+}3}\ \underline{3}\ I \to D(E(F(\lambda abc{:}I)))\ \underline{5}\ \underline{8}\ \underline{3}\ I \\
&\to E(F(\lambda abc{:}I))\ \underline{5}\ \underline{8}\ \underline{3}\ \underline{I,8} \to E(F(\lambda abc{:}I))\ \underline{5}\ \underline{8}\ \underline{3}\ \langle\underline{8}\rangle \\
&\to F(\lambda abc{:}I)\ \underline{5}\ \underline{8{-}2x3}\ \underline{3}\ \langle\underline{8}\rangle \to F(\lambda abc{:}I)\ \underline{5}\ \underline{2}\ \underline{3}\ \langle\underline{8}\rangle \\
&\to (\lambda abc{:}I)\ \underline{5}\ \underline{2}\ \underline{3}\ \langle\underline{8}\rangle,\underline{2} \to (\lambda abc{:}I)\ \underline{5}\ \underline{2}\ \underline{3}\ \langle\underline{8,2}\rangle \\
&\to I\langle\underline{8,2}\rangle \to \langle\underline{8,2}\rangle\ .
\end{aligned}
$$

## 2.11 Conditional Statements

Recall that the LC representation of a boolean expression b has the property that for all LE's p and q, we have

$$
\begin{aligned}
\{b\}\ p\ q &\to p\ , \quad \text{if b has the value } \underline{\text{true}}\ , \\
&\to q\ , \quad \text{if b has the value } \underline{\text{false}}\ .
\end{aligned}
$$

In view of the above property, we choose the representation of a two-branch conditional statement as follows:

$$
\begin{aligned}
&\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\}(v_1,\ldots,v_n) \\
&\quad \equiv \lambda\phi v_1\ldots v_n\colon \{b\}(\{S_1\}\phi v_1\ldots v_n)(\{S_2\}\phi v_1\ldots v_n) \qquad (15)
\end{aligned}
$$

For the purpose of representation, a one-branch conditional statment (an *if*-statement, in ALGOL 60 terminology) may be viewed as a two-branch conditional with a dummy or "do-nothing" statement for the second branch. When appearing in the environment $(v_1,\ldots,v_n)$, the "do-nothing" statement can obviously be represented by the LE

$$
\lambda\phi v_1\ldots v_n\colon \phi v_1\ldots v_n\ ,
$$

which reduces to I. On substituting this LE for $\{S_2\}$ in (5), we obtain

$$
\begin{aligned}
&\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\}(v_1,\ldots,v_n) \\
&\quad \equiv \lambda\phi v_1\ldots v_n\colon \{b\}(\{S_1\}\phi v_1\ldots v_n)(\phi v_1\ldots v_n)\,. \qquad (16)
\end{aligned}
$$

It must be evident that if b is a boolean expression, then for for all LE's p, q, and r, the following conversion relation holds:

$$\{b\}(pr)(qr) \;=\; \{b\}\ p\ q\ r \quad .$$

The repeated application of this relation allows us to rewrite (15) in the alternative form

$$\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n\colon \{b\}\{S_1\}\{S_2\}\phi v_1 \ldots v_n \qquad (17)$$

(Notice that although the LE's $\{S_1\}$ and $\{S_2\}$ do not contain free occurrences of $v_1,\ldots,v_n$, the LE $\{b\}$ may very well contain these; therefore, these variables cannot be dropped from (17).) On taking $S_2$ to be the dummy statement in (17), hence $\{S_2\}$ to be I, we get the following alternative to (16):

$$\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1 \ldots v_n\colon \{b\}\{S_1\}I\phi\ v_1 \ldots v_n \qquad (18)$$

Example. Suppose that the statement

$$\underline{\text{if}}\ a < b\ \underline{\text{then}}\ a := a+1\ \underline{\text{else}}\ b := a+b$$

appears in the context (a,b,c) in a program. The formulas (15) and (17) respectively yield the LE's (A) and (B) as the representations of the above statement.

(A) $\lambda\phi abc\colon \underline{a<b}\ ((\lambda\phi abc\colon \phi\ \underline{a+1}\ bc)\phi abc)((\lambda\phi abc\colon \phi a\ \underline{a+b}\ c)\ \phi abc)$

$\rightarrow \lambda\phi abc\colon \underline{a<b}\ (\phi\ \underline{a+1}\ bc)(\phi\ a\ \underline{a+b}\ c)$

$= \lambda\phi abc\colon \underline{a<b}\ (\phi\ \underline{a+1}\ b)(\phi a\ \underline{a+b})\ c$

$\rightarrow \lambda\phi ab\colon \underline{a<b}\ \ (\phi\ \underline{a+1}\ b)(\phi a\ \underline{a+b})$

(B) $\lambda\phi abc\colon \underline{a<b}\ (\lambda\phi abc\colon \phi\ \underline{a+1}\ bc)(\lambda\phi abc\colon \phi a\ \underline{a+b}\ c)\ \phi abc$

$\rightarrow \lambda\phi ab\ :\ \underline{a<b}\ (\lambda\phi ab\colon \phi\ \underline{a+1}\ b)(\lambda\phi ab\colon \phi a\ \underline{a+b})\ \phi ab$

## 2.12 Arrays

Arrays can be interpreted as tuples and hierarchies of tuples, as follows: An array of a single dimension is represented by a tuple of the representations of the individual array elements, taken in the order of the lowest to the highest subscript. An array of dimension n+1 is represented by a tuple whose elements are the representations of the n-dimensional subarrays (or slices, in the ALGOL 68 terminology [13]) obtained by fixing the first subscript in turn from the lowest to the highest possible value. For example, the array A[1:2, 1:3] is represented by

$$\langle\langle \underline{A}_{11}, \underline{A}_{12}, \underline{A}_{13}\rangle, \langle \underline{A}_{21}, \underline{A}_{22}, \underline{A}_{23}\rangle\rangle \quad ,$$

where $\underline{A}_{ij}$ is the representation of the array element A[i,j]. As in the case of simple variables, an array identifier can itself be used as the LC variable to denote the array.

With the above interpretation of arrays, we next describe the representation of subscripted variables in expressions, assignments to subscripted variables, and array declarations. In this description, we assume, for simplicity, that all arrays have the lowest subscript bound of 1.

Subscripted variable as an operand in an expression. The representation in this case is just the corresponding element of the tuple representing the array. Thus, we need some operator to extract an element of a tuple, given the element position. For given integers i and m such that $1 \leq i \leq m$, define the LE

$$\underline{\text{elem}}_{i,m} \equiv \lambda x_1 \ldots x_m : x_i \quad ,$$

and note that

$$\langle a_1, \ldots, a_m\rangle \ \underline{\text{elem}}_{i,m} \rightarrow a_i \quad .$$

Thus, given the declaration array v[1:m], we have

$$\{v[i]\} \equiv v \ \underline{\text{elem}}_{i,m} \quad .$$

This representation is inadequate, since, in general, i and m are given as expressions rather than constants, and their values may not be known at the time of LC translation of the program. However, it can be shown [11] that there is an LE elem such that for all LE's a and b, if $a \to \underline{i}$ and $b \to \underline{m}$ for integers $1 \leq i \leq m$, then

$$\underline{\text{elem}}\ a\ b \to \underline{\text{elem}}_{i,m} \ .$$

Hence, given the array declaration v[1:e], we define

$$\{v[f]\} \equiv v(\underline{\text{elem}}\ \{f\}\{e\}) \ .$$

More generally, for array $v[1:e_1,\ldots,1:e_n]$ ,

$$\{v[f_1,\ldots,f_n]\} \equiv v(\underline{\text{elem}}\ \{f_1\}\{e_1\})\ \ldots\ (\underline{\text{elem}}\ \{f_n\}\{e_n\}) \ .$$

Assignments to subscripted variables. The representation consists in replacing the designated element of the tuple representing the array with the representation of the new value, and requires changing all the slices that contain the element. As an example of a replacement operator, define, for given integers i and m such that $1 \leq i \leq m$,

$$\underline{\text{replace}}_{i,m} \equiv \lambda y x_1 \ldots x_m : \langle x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_m\rangle \ .$$

Then, it is easy to see that for all LE's $a_1,\ldots,a_m,b$,

$$\langle a_1,\ldots,a_m\rangle(\underline{\text{replace}}_{i,m}\ b) \to \langle a_1,\ldots,a_{i-1},b,a_{i+1},\ldots,a_n\rangle \ .$$

Again, one can define [11] an LE replace with the property that for all LE's a and , b, if $a \to \underline{i}$ and $b \to \underline{m}$, where i and m are integers, $1 \leq i \leq m$, then

$$\underline{\text{replace}}\ a\ b \to \underline{\text{replace}}_{i,m} \ .$$

Hence, given the declaration array $v_j[1:e]$, we have

$$\{v_j[f]:=g\}_{(v_1,\ldots,v_n)} \equiv \lambda\phi v_1\ldots v_n : \phi v_1\ldots v_{j-1}(v_j(\underline{\text{replace}}\{f\}\{e\}\{g\}))\ v_{j+1}\ldots v_n \ .$$

The above representation can be generalized to the case of arrays of dimension higher than one [11], but we omit these details.

Array declaration. Recall that a variable declaration affects the representation of the block in whose head it occurs; specifically, the block representation has as a component the initial value of the variable. Array declarations are handled precisely in the same fashion as the simple variables, with the following exception: if an array is not initialized at the time of declaration, the block representation is obtained by assuming all the array elements to be $\Omega$. Thus, for an array with the bound pairs [1:2, 1:3], the initial value is represented by $\langle\langle\Omega,\Omega,\Omega\rangle,\langle\Omega,\Omega,\Omega\rangle\rangle$. Since, in general, array bounds may be specified by expressions, we need to create tuples of arbitrary dimensions and sizes in which all elements are $\Omega$. This is possible by means of an LE tupinit with the property

$$\underline{\text{tupinit}}\ \underline{1}\ \underline{m} \rightarrow \underbrace{\langle\Omega,\Omega,\Omega\ldots,\Omega\rangle}_{m \text{ elements}}$$

$$\underline{\text{tupinit}}\ \underline{n+1}\ \underline{m}_1 \cdots \underline{m}_{n+1} \rightarrow \underbrace{\langle A,A,\ldots,A\rangle}_{m_1 \text{ elements}} \quad , \text{ where}$$

$$A \equiv \underline{\text{tupinit}}\ \underline{n}\ \underline{m}_2 \cdots \underline{m}_{n+1}\ .$$

The definition of tupinit may be found in [11].

Example. The representations discussed above are illustrated on the following block, assumed to occur in the environment (n).

```
begin array p[1:n], q[1:2, 1:2]; integer r;
   r := q[e,f];    A ≡ λφpqrn:φpq(q(elem{e}2)(elem{f}2))n
   p[r] := g       B ≡ λφpqrn:φ(p(replace r n {g})) qrn
end    C ≡ λφn:A(B(λpqr:φ))(tupinit 1 n)<<Ω,Ω>,<Ω,Ω>>Ωn
```

# 3. ITERATION AND JUMPS

## 3.1 Recursive Definition of Lambda-Expressions

In dealing with program loops, we will be in need of the recursive definitions of the form

$$A \equiv \ldots A \ldots A \ldots , \tag{19}$$

in which an LE is defined in terms of itself, by using its own name for one or more of its components. Much towards the precise understanding of such definitions has been contributed, among others, by Morris [14], Manna [3,15,16], de Bakker [17], and Rosen [18], who have built upon the pioneering work of Kleene [19] and Scott [20,21]. Referring the reader to the cited work for a rigorous analysis of recursive definitions, we shall be content here with informal, intuitive comments.

A possible explanation of (19) is the following purely mechanical one. As would be possible in the case of a non-recursive definition, we allow the replacement of an appearance of A in any LE by the right-hand side of (19). In other words, we interpret (19) as a reduction rule

$$A \rightarrow \ldots A \ldots A \ldots . \tag{19a}$$

By a sequence of such reductions of A and the other LC contractions, it may be possible to reduce an LE containing A to a normal form. It has been shown (e.g., Rosen [18]) that the Church-Rosser property holds with this broader sense of reduction also; consequently, most other important properties of reduction, such as the uniqueness of normal forms and the correctness of the standard-order reduction algorithm, are valid when the reduction rules of form (19a) (with distinct left-hand sides) are admitted.

The interpretation of a recursive definition as a replacement or reduction rule certainly enables us to use the definition; but what, actually, is the object that is so defined? It is obvious that applying the reduction rule (19a) to A itself cannot lead to an "ordinary" definition -- a non-self-referencing -- description of A. However, such a definition may be possible

if we interpret A as a solution of the reduction relation (not rule)

$$A \to \ldots A \ldots A \ldots , \tag{20}$$

or the conversion relation,

$$A = \ldots A \ldots A \ldots . \tag{21}$$

These relations have, in general, infinitely many solutions. To see this in the case of (21), rewrite it in the form

$$A = (\lambda x\colon \ldots x \ldots x \ldots)A ,$$

or

$$A = FA , \tag{22}$$

where $F \equiv \lambda x\colon \ldots x \ldots x \ldots$ does not involve A. It is now immediate that for all LE's p, the LE $F^{\infty}p$ (where $a^{n}b \equiv \underbrace{a(a(\ldots(a}_{n\text{ times}}b)\ldots))$ and $a^{\infty}b \equiv \lim_{n\to\infty} a^{n}b$) satisfies (22).

Also, note that because the LE

$$Y \equiv \lambda x\colon (\lambda y\colon x(yy))(\lambda x\colon x(yy))$$

has the property

$$Ya \to a(Ya) \tag{23}$$

for all a, the LE YF is another solution of (22).

To reduce an LE containing A, we may, under this latter interpretation of the definition (19), replace A by a solution of (21), and then apply a sequence of contractions. The results of reduction under this interpretation and the ones under the previous interpretation (of using (19) as a reduction rule) may, in general, be expected to be different.

Now let us define a partial order on LE's as follows: $a \leq b$ (a is extended by b), if for all c it is the case that whenever ca has a normal form, then ca = cb. For example, we have $\Omega \leq b$ for all LE's b, where $\Omega$ is as introduced in Sect. 2.8. One can show [14] that $F^{\infty}\Omega$, as well as YF, are minimal solutions of (22), that is, they are extended by all LE's satisfying that relation. It turns out that, for all LE's containing A, the normal form reductions resulting from the

replacement of A by a minimal solution of (21) in the second interpretation are precisely the same as those resulting under the first interpretation of using (19a) as a reduction rule. It is natural, therefore, to identify the LE A defined by (19) with a minimal solution of (21). Although, in general, (21) has infinitely many, mutually non-convertible, minimal solutions, these solutions are equivalent in the sense that they are all extended by each other, and have the same intuitive interpretation as functions. Hence, we arbitrarily adopt one of these, YF, with F as defined earlier, as the minimal solution.

Sometimes, minimal solutions are expressed in the "μ-notation" [17]: we write the minimal solution of (21) as the μ-expression

$$\mu x: \ldots x \ldots x \ldots \ .$$

The μ-expressions resemble λ-expressions in variable binding properties; and they have the associated rules

$$\mu x: e \to \mu y: (\lambda x{:}e)y \tag{24}$$

$$\mu x: e \to (\lambda x{:}e)(\mu x{:}e) \tag{25}$$

The first rule simply renames the "μ-bound" variable, while the second becomes obvious in view of the fact that being a solution of $x = e = (\lambda x{:}e)x$, the LE $\mu x{:}e$ must itself satisfy this conversion relation for x.

The above interpretations can also be generalized to include the simultaneous recursive definition of several LE's in the form

$$\begin{aligned} A_1 &\equiv F_1 A_1 \ldots A_n \ , \\ &\ldots \\ A_n &\equiv F_n A_1 \ldots A_n \ . \end{aligned} \tag{26}$$

where, it is understood, the LE's F's do not involve A's. The generalized interpretations are, briefly:

(a) The definitions (26) may be regarded as a set of reduction rules (replacing ≡ by → ) without concern as to the values of A's.

(b) The A's defined by (26) may be regarded to be the

minimal solutions of the system of conversion relations

$$A_i = F_i A_1 \cdots A_n \ , \qquad 1 \leq i \leq n \ . \tag{27}$$

An explicit soluion of (27) is given by

$$A_i \equiv Y_{i,n} \, F_1 \cdots F_n \ ,$$

where

$$Y_{i,n} \equiv \lambda z_1 \cdots z_n \colon Y(\lambda x \colon \langle xz_1, \ldots, xz_n \rangle)(\lambda x_1 \cdots x_n : x_i) \ .$$

Alternatively:

$$A_i \equiv A_i^{(\infty)} \ ,$$

where

$$A_i^{(0)} \equiv \Omega \ , \quad \text{and} \quad A_i^{(j+1)} \equiv F_i A_1^{(j)} \cdots A_n^{(j)} \ .$$

The interpretations (a) and (b) result in identical normal form reductions of the LE's containing A's.

## 3.2. Iteration Statements

The representation of the for statement of ALGOL 60 is obtained by expressing this statement in terms of the simple (non-ALGOL 60) while loop of the form while ... do ... . To represent the latter, consider the statement while b do S appearing in the environment $(v_1, \ldots, v_n)$. Calling this statement by the name T, we may (recursively!) describe it, for the purpose of LC representation, as

if b then begin S;T end .

Now the formulas for the representation of compound and conditional statements, (5) and (16), respectively, are applicable to the above statement, so that its representation {T} is, recursively, the LE

$$\lambda \phi v_1 \cdots v_n \colon \{b\}((\lambda \phi \colon \{S\}(\{T\}\phi))\phi v_1 \cdots v_n)(\phi v_1 \cdots v_n)$$
$$= \lambda \phi v_1 \cdots v_n \colon \{b\}(\{S\}(\{T\}\phi))\phi v_1 \cdots v_n \ .$$

Thus, we adopt the representation

$$\{\underline{\text{while}}\ b\ \underline{\text{do}}\ S\}_{(v_1,\ldots,v_n)} \equiv \mu x\colon \lambda\phi v_1\ldots v_n\colon \{b\}(\{S\}(x\phi))\phi v_1\ldots v_n. \tag{28}$$

Alternative definitions of the same LE, call it A, are

$$A \equiv \lambda\phi v_1\ldots v_n\colon \{b\}(\{S\}(A\phi))\phi v_1\ldots v_n\ ,$$

$$A \equiv Y(\lambda x\phi v_1\ldots v_n\colon \{b\}(\{S\}(x\phi))\phi v_1\ldots v_n)\ .$$

Example. At this point, we illustrate the LC representations introduced so far by means of a complete program. Also, as an application of the model, we derive the correctness of the program in terms of its representation. Given below are the individual statement representations, shown on the same line as the statements (or on the last line for multiple line statements), and have been designated names for reference purposes.

```
begin integer x, y;
   read x;                          A ≡ λφxyoi:φiyo
   y := 0;                          B ≡ λφxy:φxO
   begin integer z;
      z := 0;                       C ≡ λφzxy:φOxy
      while z < x do
         begin
            y := 1+y+2× z;          D ≡ λφzxy:φzx 1+y+2×z
            z := z+1                E ≡ λφzxy:φ z+1 xy
         end                        F ≡ λφ: D(Eφ)
      end while                     G ≡ λφzxy: z<x (F(Gφ))φzxy
   end;                             H ≡ λφ: C(G(λz:φ))Ω
   write y                          J ≡ λφxyo:φxyo,y
end                                 K ≡ λφ: A(B(H(J(λxy:φ))))ΩΩ
                    {program} ≡ P ≡ KII
```

We wish to prove that on reading a nonnegative integer n, this program will print out the integer $n^2$. According to our input-output conventions, we need to show that

$$P\ \underline{n} \rightarrow \langle \underline{n^2} \rangle \ , \quad \text{for all integers } n \geq 0 \ . \tag{i}$$

This is done in four steps, as follows.

(a) We show that for all LE $\phi$ and all integers n and i,

$$G\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \ , \qquad \text{if } i \geq n \ , \tag{ii}$$

$$G\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow G\phi\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2} \ , \quad \text{if } i < n \ . \tag{iii}$$

By using the definition of G, we obtain

$$G\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow \underline{i < n}\ (F(G\phi))\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2}$$

If $i \geq n$, then $\underline{i < n} \rightarrow \underline{\text{false}}$, so that (ii) is immediate. Otherwise, $\underline{i < n} \rightarrow \underline{\text{true}}$, and the above LE

$$\rightarrow F(G\phi)\underline{i}\ \underline{n}\ \underline{i^2} \rightarrow D(E(G\phi))\underline{i}\ \underline{n}\ \underline{i^2}$$

$$\rightarrow E(G\phi)\underline{in}\ \underline{1+i^2+2\times i} \rightarrow E(G\phi)\ \underline{i}\ \underline{n}\ \underline{(i+1)^2}$$

$$\rightarrow G\phi\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2} \ .$$

(b) Next, for all integers n and i such that $0 < i \leq n$, we have

$$G\ \phi\ \underline{0}\ \underline{n}\ \underline{0} \rightarrow G\ \phi\ \underline{i}\ \underline{n}\ \underline{i^2} \ . \tag{iv}$$

This is proved by induction on i. From (iii) one easily verifies (iv) both for $i = 1$, and for $i = j+1 \leq n$ when the case for $i = j < n$ is assumed.

(c) Next, we claim that for all integers $n \geq 0$, it is the case that

$$H\ \phi\ \underline{n}\ \underline{0} \rightarrow \phi\ \underline{n}\ \underline{n^2} \ . \tag{v}$$

For, we have

$$H\phi\underline{n}\underline{0} \equiv (\lambda\phi\colon C(G(\lambda z{:}\phi))\Omega)\phi\ \underline{n}\ \underline{0}$$

$$\rightarrow C(G(\lambda z{:}\phi))\Omega\ \underline{n}\ \underline{0}$$

$$\rightarrow G(\lambda z{:}\phi)\underline{0}\ \underline{n}\ \underline{0}$$

Now if $n = 0$, then from (ii) it follows that

$$G(\lambda z{:}\phi)\underline{0}\ \underline{n}\ \underline{0} \rightarrow (\lambda z{:}\phi)\underline{n}\ \underline{n}\ \underline{n^2} \rightarrow \phi\ \underline{n}\ \underline{n^2}\ .$$

On the other hand, if $n > 0$, then for the case $i = n$, (iv) yields

$$G(\lambda z{:}\phi)\underline{0}\ \underline{n}\ \underline{0} \rightarrow G(\lambda z{:}\phi)\underline{n}\ \underline{n}\ \underline{n^2}$$

$$\rightarrow (\lambda z{:}\phi)\underline{n}\ \underline{n}\ \underline{n^2}\ , \quad \text{by (ii)}$$

$$\rightarrow \phi\ \underline{n}\ \underline{n^2}\ .$$

(d) Finally, to prove (i) we simply use the definitions of the LE's A through P, obtaining, for all integers $n \geq 0$,

$$P\ \underline{n} \equiv K\ I\ I\ \underline{n} \rightarrow A(B(H(J(\lambda xy{:}I))))\Omega\Omega I\underline{n}$$

$$\rightarrow B(H(J(\lambda xy{:}I)))\underline{n}\ \Omega\ I$$

$$\rightarrow H(J(\lambda xy{:}I))\underline{n}\ \underline{0}\ I$$

$$\rightarrow J(\lambda xy{:}I)\underline{n}\ \underline{n^2}\ I\ , \quad \text{by (v)}$$

$$\rightarrow (\lambda xy{:}I)\underline{n}\ \underline{n^2}\ \underline{I,n^2}$$

$$\rightarrow \underline{I,n^2}$$

$$\rightarrow \langle\underline{n^2}\rangle\ .$$

Returning to the discussion of iteration statements, we can express the general for statement of ALGOL 60 in terms of the simple while loop treated above. For example, we can reformulate the statement

$$\underline{\text{for}}\ v_i := e_1\ \underline{\text{step}}\ e_2\ \underline{\text{until}}\ e_3\ \underline{\text{do}}\ S$$

as

$$\underline{\text{begin}}\ v_i := e_1;\ \underline{\text{while}}\ (v_i - e_3) \times \text{sign}(e_2) \leq 0\ \underline{\text{do}}$$
$$\underline{\text{begin}}\ S;\ v_i := v_i + e_2\ \underline{\text{end}}\ \underline{\text{end}}\ .$$

The latter form can then be represented as an LE by employing the representations of compound and while statements. Omitting the details of derivation, we list below the LC representations for the three cases of for list elements, namely, arithmetic expression, while element, and step-until element:

$$\{\underline{\text{for}}\ v_i = e\ \underline{\text{do}}\ S\}_{(v_1,\ldots,v_n)}$$
$$\equiv \lambda\phi v_1 \ldots v_n\colon \{S\}\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n\ . \tag{29}$$

$$\{\underline{\text{for}}\ \ v_i := e\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\}_{(v_1,\ldots,v_n)}$$
$$\equiv \mu x{:}\lambda\phi v_1 \ldots v_n{:}(\lambda v_i{:}\{b\})\{e\}(\{S\}(x\phi))\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n \tag{30}$$
$$\equiv Y(\lambda x\phi v_1 \ldots v_n{:}(\lambda v_i{:}\{b\})\{e\}(\{S\}(x\phi))\phi v_1 \ldots v_{i-1}\{e\}v_{i+1}\ldots v_n)$$

$$\{\underline{\text{for}}\ v_i \equiv e_1\ \underline{\text{step}}\ e_2\ \underline{\text{until}}\ e_3\ \underline{\text{do}}\ S\}_{(v_1,\ldots,v_n)}$$
$$\equiv \lambda\phi v_1 \ldots v_n{:}(Y(\lambda x\phi v_1 \ldots v_n{:}\{(v_i - e_3)\times \text{sign}(e_2) \leq 0\}$$
$$(\{S\}(\lambda v_1 \ldots v_n{:}x\phi v_1 \ldots v_{i-1}\{v_i + e_2\}v_{i+1}\ldots v_n))$$
$$\phi v_1 \ldots v_n))\phi v_1 \ldots v_{i-1}\{e_1\}v_{i+1}\ldots v_n\ . \tag{31}$$

## 3.3 Jumps and Labels

To execute the statement S ≡ goto L, we may, in effect, substitute the part of the program following L for the one following S. This observation is the basis of our representation of both labels and jump statements.

A label is identified with the part of the program following it. To be accurate, the representation of a label L occurring in a program P is taken to be the representation of the program P' obtained from P by deleting all the statements, but retaining the declarations, that appear above L. This representation can be obtained in a simpler manner by using the following inductive scheme: Let the label L occur in a block b whose declared variables are $v_1,\ldots,v_n$.

1) If L is followed by statements $S_1,\ldots,S_m$, and a label M, in that order, all within b, then

$$\{L\} \equiv \{S_1\}(\{S_2\}(\ldots(\{S_m\}\{M\})\ldots))$$

2) If $S_1,S_2,\ldots,S_m$ are the statements following L to the end of b, then

$$\{L\} \equiv \{S_1\}(\{S_2\}(\ldots(\{S_m\}(\lambda v_1\ldots v_n{:}N))\ldots)) \ ,$$

where N ≡ I, if b is the outermost block, else N is the representation of the program part following b, that is, of the (possibly imaginary) label immediately after the end of b.

According to the rules of ALGOL, the label to which a jump can be made must be in a block which is the same as, or outer to, the block containing the jump statement. It follows that (the list of variables constituting) the environment of a jump statement must contain the environment of the referred label as a final segment. Suppose $(v_1,\ldots,v_n)$ is the environment of the statement S ≡ goto L, and $(v_m,\ldots,v_n)$, where $1 \leq m \leq n$, is the environment of L, and let $\phi$ represent as usual the program

remainder of S. The execution of S causes the program to compute the function $\{L\}(v_m,\ldots,v_n)$ instead of $\phi(v_1,\ldots,v_m)$. Hence, the representation of S can be taken to be the LE

$$\lambda\phi v_1\ldots v_n\colon\ \{L\}v_m\ldots v_n\ ,$$

which reduces to

$$\lambda\phi v_1\ldots v_{m-1}\colon\ \{L\}\ .$$

Thus, we define

$$\{\underline{\text{goto}}\ L,\ \text{environment}(L) = (v_m,\ldots,v_n),\ 1 \le m \le n\}_{(v_1,\ldots,v_n)}$$

$$\equiv \lambda\phi\colon\ (\lambda v_1\ldots v_{m-1}\colon\{L\})\ . \qquad (32)$$

It is sometimes convenient, specially in connection with conditional statements, to write the right-hand side LE in the convertible forms

$$\lambda\phi v_1\ldots v_n\colon\ \{L\}v_m\ldots v_n \qquad \text{or} \qquad \lambda\phi v_1\ldots v_n\colon\ (\lambda v_1\ldots v_{m-1}\colon\{L\})v_1\ldots v_n.$$

Example. The representation of goto statements and labels is illustrated by means of a complete program. The program below has been derived from the program given in the previous example simply by expressing the while loop in terms of goto's. As another application of the model, we prove the (input-output) equivalence of the two programs.

As before, the representations of individual statements are shown on the same line as the statement, or on the last line for a multiple-line statement, and are designated identifying names. The LE's common to the representation of both programs have the same names. Label representations are given at the end. (The superfluous label M serves simply to illustrate the case (1) of label representations discussed above.)

```
begin integer x,y;
   read x;                          A ≡ λφxyoi:φiyo
   y := 0;                          B ≡ λφxy:φx0
   begin integer z;
      z := 0;                       C ≡ λφzxy:φ0xy → λφz: φ0
L:    if z=y then goto N
            else goto M;            D'≡ λφzxy: z=y (λz:N)Mzxy
M:    y := y+2×z+1;                 E'≡ λφzxy:φzx y+2×z+1
      z := z+1;                     F'≡ λφzxy:φ z+1 xy
      goto L                        G'≡ λφ: L
   end;                             H'≡ λφ: C(D'(E'(F'(G'(λz:φ)))))Ω
N: write y;                         J ≡ λφxyo:φxy o,y
end                                 K'≡ λφ: A(B(H'(J(λxy:φ))))ΩΩ
                    {Program} ≡ P' ≡ K'II
```

$$L \equiv D'M\ , \quad M \equiv E'(F'(G'(\lambda z{:}N)))\ , \quad N \equiv J(\lambda xy{:}I)$$

We wish to prove that the above program and the program of the previous example produce the same output when executed with the same non-negative integer as the input datum. That is, in terms of their representations, we wish to show that for all integers $n \geq 0$,

$$P\ \underline{n} = P'\underline{n}\ . \qquad \text{(i)}$$

Of course, this can be shown by using the previously obtained result $P\underline{n} \rightarrow \langle \underline{n^2} \rangle$ in conjunction with a direct proof of the fact that $P'\underline{n} \rightarrow \langle \underline{n^2} \rangle$. But we will prove the equivalence of the programs by verifying, in effect, that their differing parts do the same work when the programs are executed. These differing parts are represented by the LE's H and H'. If we can show that for all integers $n \geq 0$,

$$H\ N\ \underline{n}\ \underline{0} = H'N\ \underline{n}\ \underline{0} \qquad \text{(ii)}$$

(where $N \equiv J(\lambda xy{:}I)$, defined in the present example), then (i) is demonstrated as follows. From the previous example, part (d), we know that for all $n \geq 0$,

$$P\ \underline{n} \rightarrow H(J(\lambda xy{:}I))\underline{n}\ \underline{0}\ I$$

$$\equiv H\ N\ \underline{n}\ \underline{0}\ I\ .$$

Since P and P' differ only with respect to their components H and H', respectively, we must also have for all $n \geq 0$,

$$P'\underline{n} \rightarrow H'N\ \underline{n}\ \underline{0}\ I\ .$$

Hence, $P\ \underline{n} = P'\underline{n}$ by (ii).

It remains to verify (ii). From (v) in the previous example, we have for all integers $n \geq 0$,

$$H\ N\ \underline{n}\ \underline{0} \rightarrow N\ \underline{n}\ \underline{n^2}\ .$$

So (ii) would follow if we can also prove

$$H'N\ \underline{n}\ \underline{0} \rightarrow N\ \underline{n}\ \underline{n^2}\ . \qquad \text{(iii)}$$

To outline the proof of (iii), we simply state the sequence of reduction relations leading to it.

(1) $$L\ \underline{i}\ \underline{n}\ \underline{i^2} \rightarrow \begin{cases} N\ \underline{n}\ \underline{n^2} & , \text{ if } i = n\ , \\ L\ \underline{i+1}\ \underline{n}\ \underline{(i+1)^2} & , \text{ if } i \neq n\ . \end{cases}$$

(2) $$L\ \underline{0}\ \underline{n}\ \underline{0} \rightarrow L\ \underline{i}\ \underline{n}\ \underline{i^2}\ , \qquad \text{for } 0 < i \leq n$$

(3) $$L\ \underline{0}\ \underline{n}\ \underline{0} \rightarrow N\ \underline{n}\ \underline{n^2}\ , \qquad \text{for } n \geq 0$$

(4) $$H'\phi\ \underline{n}\ \underline{0} \rightarrow N\ \underline{n}\ \underline{n^2}\ , \qquad \text{for } n \geq 0\ .$$

The treatment of designational expressions and switches is omitted, except for an example which should suffice to indicate how these can be represented as LE's. In the schematic program below, b and c denote Boolean, and e and f, arithmetic expressions.

```
begin integer x;
   M: ...
   begin integer y;
      ...
      begin integer z;
         switch S := K, if b then S[e] else L, M;
         ...
         K: ...
         begin integer w;
            ...
            goto if c then M else S[f];
         end w;
      end z;
      L: ...
   end y;
end
```

The representations of the switch and goto statements in the above program are, respectively,

$$\underline{S} \equiv \langle \underline{K},\ \underline{b}(\lambda zyx:\ \underline{S}[\underline{e}]_3 zyx)(\lambda z{:}\underline{L}),\ \lambda zy{:}\underline{M} \rangle \quad ,$$

$$\lambda \phi wzyx:\ \underline{c}(\lambda wzy{:}\underline{M})(\lambda w:\ \underline{S}[\underline{f}]_3) \quad ,$$

where, for brevity, underlined letters are used to designate corresponding representations, and the LE (elem i n) (Cf. Sec. 2.12) is written $[\underline{i}]_n$.

# 4. PROCEDURES

## 4.1 Functions

We use the term function to denote a type procedure without any side effects. In particular, a function is a procedure in which

(1) a value is associated with the procedure name,

(2) all parameters are called by value,

(3) no global variables are modified,

(4) no jumps are made outside the function body, and no procedures are used other than functions.

Because of the above restrictions, the representation of functions is much simpler than that of general procedures. Since many procedures encountered in programs are truly functions, it seems useful to deal with them as a special case.

For the moment, let us consider only the functions which do not involve global variables at all. For these, the environment of the function declaration is immaterial. Let $f(p_1,\ldots,p_n)$ be such a function with parameters $p_i$. We wish to represent $f$ in such a manner that for all expressions $e_1,\ldots,e_n$

$$\{f\}\{e_1\}\ldots\{e_n\} \rightarrow \underline{f(e_1,\ldots,e_n)} \qquad \text{(i)}$$

Such a representation is accomplished as follows: We use a variable $\pi$ to denote the function value; that is, all assignments to $f$ are represented as if made to $\pi$. Further, we represent the statement $S$ constituting the body of $f$ by taking its environment to be $(\pi,p_1,\ldots,p_n)$. Now, starting with an arbitrary value of $\pi$, and the values $e_i$ of $p_i$, the execution of $S$ has the effect of assigning the value $f(e_1,\ldots,e_n)$ to $\pi$, and certain values to $p_i$ which are irrelevant to the result; say, we have

$$\{S\}\phi\pi\{e_1\}\ldots\{e_n\} \rightarrow \phi\ \underline{f(e_1,\ldots,e_n)}\ p_1\ldots p_n\ . \qquad \text{(ii)}$$

Comparing (i) and (ii), we can choose the LE $\lambda\pi p_1 \ldots p_n{:}\pi$ for $\phi$, $\Omega$ for $\pi$, and $\{S\}\phi\pi$ for $\{f\}$, so as to satisfy (i). Thus, we adopt

$$\{\underline{\text{function}}\ f(p_1,\ldots p_n)\ \text{with body}\ S\} \equiv \{S\}_{(\pi,p_1,\ldots p_n)}(\lambda\pi p_1\ldots p_n{:}\pi)\Omega. \tag{33}$$

It should be pointed out that a label appearing in the body of a function is to be represented as the part of the function (not the program) that follows the label.

Example. The following procedure is a function; hence (33) is applicable.

```
integer procedure mod(x,y); value x,y; integer x,y;
   begin integer q;
      q := x÷y;
      mod := x-y×q
   end q
end mod;
```

$A \equiv \lambda\phi q\pi xy{:}\phi\ \underline{x\div y}\ \pi xy$

$B \equiv \lambda\phi q\pi xy{:}\phi q\ \underline{x-y\times q}\ xy$

$C \equiv \lambda\phi{:}\ A(B(\lambda q{:}\phi))\Omega$

$\underline{\text{mod}} \equiv C(\lambda\pi xy{:}\pi)\Omega$

Example. Simplification of a function representation.

```
integer procedure fact(n); value n; integer n;
   fact := if n ≤ 0 then 1 else n × fact(n-1);
```

In this case, we have

$\{\text{body}\} \equiv \lambda\phi\pi n{:}\ \phi(\underline{n=0}\ \underline{1}\ \underline{(n \times (\underline{\text{fact}}(n-1)))}n$

$\underline{\text{fact}} \equiv \{\text{body}\}(\lambda\pi n{:}\pi)\Omega$

$\rightarrow \lambda n{:}\ \underline{n=0}\ \underline{1}\ \underline{(n\times(\text{fact}(n-1)))},$

so that, alternatively,

$\underline{\text{fact}} \equiv Y(\lambda zn{:}\ \underline{n=0}\ \underline{1}\ \underline{(n\times(z(n-1)))}\ )\ .$

Finally, it is easy to lift the restriction about global variables imposed earlier on functions: All one needs is to treat each variable in the environment of a function declaration as a parameter, in addition to the explicitly declared parameters of the function. This is illustrated below.

Example.

```
begin integer x,y;
   ...
   integer procedure f(n); value n; integer n;
      f := n+x;                    A ≡ λφπnxy:φ n+x nxy
   ...                             f ≡ A(λπnxy:π)Ω
   begin integer z;
      x := f(y)+z                  λφzxy:φz(fyxy)+z y
...
```

## 4.2 Call-by-name, Side-effects

In the previous section, we have described the LC representation of procedures subject to rather stringent conditions. We will now show how the representations can be extended to more general procedures, allowing call-by-name, the use of global variables, and side effects. However, we limit ourselves here to considering the formal paramters of the type integer and label only. But the extension of the model to include real and boolean parameters is trivial.

In ALGOL 60, a procedure call is intended to have the effect of an appropriately modified copy of the procedure body [12]. The modification in the case of call-by-name consists in replacing each instance of a called-by-name formal parameter by the corresponding actual parameter. (It is understood that any name conflicts between the variables appearing in the actual parameter expressions and the local variables of the procedure are to be first removed by renaming the latter variables.) Instead of performing such symbolic substitution, however, which would require

keeping procedures in text form at the execution time, ALGOL compilers accomplish the same effect by treating formal parameter references in procedure as calls on special "parameter procedures" generated from actual parameters [22]. As a result, if an operation refers to a formal parameter during the execution of a procedure, then the procedure execution is suspended to evaluate the corresponding actual parameter in the environment of the procedure calling statement, and then the procedure execution is resumed using the so-acquired value in the operation. Of course, depending upon the type and use of a parameter, the actual parameter evaluation may yield a value (e.g., an arithmetic or Boolean quantity when the formal parameter is an operand in an expression) or a name (e.g., the address of a variable when the formal parameter appears to the left of an assignment statement.) Our LC interpretation is based on a similar idea. But we make use of different "parameter procedures" for different operations performed with the same parameter; namely, the evaluation of actual parameter expressions, making assignments to the variables provided as actual parameters, and jump to an actual label.

## 4.3. Integer Parameters

In the absence of procedures we were able to express each statement in a program as a function which had for its arguments the variable $\phi$, denoting the program remainder (that is, the part of the program following the statement), and the variables constituting the environment of the statement. Clearly the representation of a statement S in a procedure body would involve two sets of program remainders and environments -- namely, one set for S itself and one for the statement, say T, that calls the procedure. The program remainder of T corresponds to the familiar "return" address or label for the procedure call. Now, any formal parameter instances in S give rise to actual parameter evaluations in the environment of T, but after the evaluation the control must eventually transfer back to S.

Hence the representation of parameter evaluation also involves the two sets of environments and program remainders; but this time the program remainder of S serves as the return address. We will use the variable $\rho$ to indicate the program remainder at the return point and $\phi$, as usual, for the program remainder at the current point.

We have so far represented, and will continue to represent, each program variable by a single LC variable. The representation of an assignment statement may be conceived as "binding" the LC variable representing the variable appearing at the left-hand side to the representation of the right-hand expression.[1] In general, the LC variables representing program variables are "bound" at any time to the current values of the corresponding program variables. With each called-by-value formal parameter we similarly need to associate a single LC variable, bound to the current "value" of the parameter at any time. However, we need to carry more information with a called-by-name formal parameter; depending on the type and use of a parameter we shall associate a number of LC variables with it. For each called-by-name formal parameter of type integer, we require three variables best thought of as being bound, respectively, to the "value" associated with it and to the "parameter procedures" for evaluating it and making assignments to it. If p is an integer parameter, then these three variables will be usually denoted by p, $p_\varepsilon$, and $p_\alpha$. (The parameter of type label will be discussed later.) The environment of a statement in a procedure body will consist of the following, in the given order:

a) variables local to the procedure,

b) $\rho$, the return variable,

c) variables representing the formal parameters,

d) variables global to the procedure.

---

[1] The present descriptive use of "binding" and "bound" has no connection with the terms defined at the beginning of Sec. 2.1.

Next, associated with each called-by-name actual parameter p of type integer, and individual to each procedure call, is an LE that represents the parameter procedure for its evaluation; in case p is a program variable (rather than an expression), there is also another LE which represents the "parameter procedure" to effect assignments to p called for in the procedure. These "actual evaluation" and "actual assignment" operators are usually denoted $\varepsilon_p^a$ and $\alpha_p^a$, respectively, with further distinguishing marks added when more than one procedure call is involved.

Lastly, associated with each called-by-name formal parameter of type integer, and unique to each environment within the procedure body, are two LE's which represent the calls on the "actual evaluation" and "actual assignment" parameter procedures mentioned above. For convenience, these LE's are referred to as "formal evaluation" and "formal assignment" operators, and are usually denoted $\varepsilon_p^f$ and $\alpha_p^f$ where p is the formal parameter, with further distinguishing marks added if more than one environment is involved. If, in a statement in a procedure body, a formal parameter appears as an operand of an expression, the statement will be represented as if preceded by a formal evaluation; likewise, if a formal parameter occurs at the left-hand side of an assignment statement, that statement will be represented as if immediately followed by a formal assignment.

The above ideas will now be illustrated by means of a very simple example:

```
begin integer a;
   procedure P(x); integer x; x:=x+2;
   ...
   P(a);
   ...
end .
```

The body of the above procedure consists of a single statement, and that statement needs to be both preceded by a formal evaluation and followed by a formal assignment. Thus, it is represented by the compound

$$\lambda\phi\colon \varepsilon_x^f(A(\alpha_x^f\phi)) \equiv B \ ,$$

say, where A is the representation of x:= x+2 as an ordinary assignment statement. Since there are no local variables in the procedure, the environment of this latter statement consists of the following:

| | |
|---|---|
| $\rho$ | the "return" variable, |
| $x_\varepsilon$ | the "parameter evaluation" variable, |
| $x_\alpha$ | the "parameter assignment" variable, |
| x | the "parameter" variable, and |
| a | the global variable. |

Thus we can write

$$A \equiv \lambda\phi\rho x_\varepsilon x_\alpha x\ a\colon \phi\rho x_\varepsilon x_\alpha \underline{x+2}\ a \ .$$

Now, as the variable $x_\varepsilon$ is bound to the "actual evaluation" operator, and the "formal evaluation" consists of just an application of this LE, we define $\varepsilon_x^f$ to be

$$\lambda\phi\rho x_\varepsilon x_\alpha x a\colon x_\varepsilon\rho\phi x_\varepsilon x_\alpha x a \ ,$$

or more simply,

$$\lambda\phi\rho x_\varepsilon x_\alpha x \ : x_\varepsilon\rho\phi x_\varepsilon x_\alpha x \ .$$

Note the interchange of $\phi$ and $\rho$ above; this signifies that the program remainder at the return point of procedure call becomes the current program remainder during parameter evaluation, and vice versa. In a similar manner, we define

$$\alpha_x^f \equiv \lambda\phi\rho x_\varepsilon x_\alpha x \ : x_\alpha\rho\phi x_\varepsilon x_\alpha x \ .$$

(In general, the global variables of the procedure need not appear in the formal evaluation and assignment operators.)

The whole procedure may be represented by

$$P \equiv B(\lambda\rho x_\varepsilon x_\alpha x\colon\rho) \ ,$$

which displays the effect that once the procedure execution is over, (after the application of B), only the return variable

is retained, and the other variables, namely, the ones connected with parameters, are deleted from the environment.

Next, let us look at the procedure call. There is only one called-by-name actual parameter of type integer in this case. So we need to define two LE's $\varepsilon_x^a$ and $\alpha_x^a$, the actual evaluation and assignment operators. These serve essentially as the fictitious assignment operators x:=a and a:=x, repsectively, and thus can be defined by

$$\varepsilon_x^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha x a\colon\ \rho\phi x_\varepsilon x_\alpha a a \ ,$$

$$\alpha_x^a \equiv \lambda\phi\rho x_\varepsilon x_\alpha x a\colon\ \rho\phi x_\varepsilon x_\alpha x x \ .$$

Again the interchange of $\phi$ and $\rho$ is needed to represent the fact that after evaluating the actual parameter in the environment of the procedure calling statement, the control passes back to the procedure body.[1]

The purpose of the procedure calling statement itself is threefold:

a) to extend the environment from (a) to $(x_\varepsilon, x_\alpha, x, a)$

b) to initialize the added variables; that is, substitute $\varepsilon_x^a$ for $x_\varepsilon$ , $\alpha_x^a$ for $x_\alpha$ , and, by convention, $\Omega$ for x.

c) to apply P before the rest of the program; that is, substitute $P\phi$ for $\phi$.

Consequently, the statement P(a) above may be represented by the LE

$$(\lambda x_\varepsilon x_\alpha x\colon\ (\lambda\phi a\colon\ P\phi x_\varepsilon x_\alpha x a))\varepsilon_x^a\alpha_x^a\Omega \ ,$$

which simplifies to

$$\lambda\phi a\colon\ P\phi\varepsilon_x^a\alpha_x^a\Omega a \ .$$

---

[1] It should not be difficult to see that coroutines can be represented by using the same idea, as follows: The "remainder" of each coroutine may be represented by a different variable. The coroutine calls are then representable by LE's which simply permute these variables to bring the remainder of the called coroutine in front. We will soon see how we can also account for the private variables of a coroutine by "covering" them when the control passes out of it and "uncovering" them on return.

We now list the representations discussed piecemeal above along with the original program.

Example.

```
begin integer a;
   procedure P(x); integer x;
                                  ε^f_x ≡ λφρx_ε x_α x: x_ε ρφx_ε x_α x
                                  α^f_x ≡ λφρx_ε x_α x: x_α ρφx_ε x_α x
      x := x+2                    A     ≡ λφρx_ε x_α xa: φρx_ε x_α x+2 a
                                  B     ≡ λφ: ε^f_x (A(α^f_x φ))
   ;                              P     ≡ B(λρx_ε x_α x:ρ)
   ...
   P(a);                          C     ≡ λφa: Pφε^a_x α^a_x Ωa
   ...                            ε^a_x ≡ λφρx_ε x_α xa: ρφx_ε x_α aa
                                  α^a_x ≡ λφρx_ε x_α xa: ρφx_ε x_α xx
end
```

Next, let us consider the LC representation of type procedures in which a value is associated with the procedure identifier. In this case we will use an additional variable $\pi$ to denote the procedure value in representing the statements of the procedure body. The representations are otherwise similar to that for routine type procedures discussed above. A procedure call which uses the function designator of a procedure within an expression will be represented as if it were compounded of two statements -- the first a procedure call to obtain the value of the procedure, and the second using that value in the expression.

The representation of a type procedure is shown in the following example, which also illustrates the treatment of call-by-value in our present scheme of procedure representation. (Some explanations follow the program.)

Example

```
begin integer u,v;
   integer procedure P(x,y); integer x,y; value y;
      begin
         P := x-y;
         x := y
      end compound
   end P;
   ...
   u := P(v,u+1) + u;
end
```

$$\varepsilon_x^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy:\ x_\varepsilon\rho\phi\pi x_\varepsilon x_\alpha xy$$

$$\alpha_x^f \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xy:\ x_\alpha\rho\phi\pi x_\varepsilon x_\alpha xy$$

$$A \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv:\ \phi\rho\ \underline{x-y}\ x_\varepsilon x_\alpha xyuv$$

$$B \equiv \lambda\phi:\ \varepsilon_x^f(A\phi)$$

$$C \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv:\ \phi\rho\pi x_\varepsilon x_\alpha yyuv$$

$$D \equiv \lambda\phi:\ C(\alpha_x^f\phi)$$

$$E \equiv \lambda\phi:\ B(D\phi)$$

$$P \equiv E(\lambda\rho\pi x_\varepsilon x_\alpha xy:\ \rho\pi)$$

$$F \equiv \lambda\phi uv:\ P\phi\Omega\varepsilon_x^a\alpha_x^a\Omega\ \underline{u+1}\ uv$$

$$\varepsilon_x^a \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv:\ \rho\phi\pi x_\varepsilon x_\alpha vyuv$$

$$\alpha_x^a \equiv \lambda\phi\rho\pi x_\varepsilon x_\alpha xyuv:\ \rho\phi\pi x_\varepsilon x_\alpha xyux$$

$$G \equiv \lambda\phi\pi uv:\ \phi\ \underline{\pi+u}\ v$$

$$H \equiv \lambda\phi:\ F(G\phi)\ .$$

The environment of the statements in the procedure above consists of eight variables: namely, the return variable $\rho$, the procedure value variable $\pi$, the three variables $x_\varepsilon$, $x_\alpha$, and x for the called-by-name parameter x, the single called-by-value parameter variable y, and finally the two global variables u and v. Of these the four parameter variables are effectively discarded at the end of the procedure body execution by the representation P of the procedure. The procedure call is represented as the compound of two statements F and G: F computes $\pi$, the procedure value, and G makes use of this in the assignment statement.

In both previous examples, the environment of the procedure declaration and the procedure call are the same. In the general case, these environments may be different; this is so, for example, when a procedure call takes place in a block inner to the procedure declaration. When this happens, there arises the problem of "covering" the local variables of the calling point that do not have valid declarations in the procedure body. Of course, the covering should be such that the variables may be "uncovered" on return to the calling point. Notice the contrast with jumps in which the variables that do not have valid declarations at the jump label are simply discarded permanently. Covering is also needed in specifying the formal evaluation and assignment operators for use with statements inside a block in a procedure body, since in this case, again, the variables local to the procedure body are invisible at the calling point.

The following example shows a way of covering the non-overlapping parts of the environment, in order to overcome the environment conflict problem. (See explanations below.)

Example

```
begin integer x;
      procedure P(y); integer y;
         begin integer z;
            ...                          ε_y^f ≡ λφzρy_ε y_α y: ρy_ε (λψ:ψφz) y_ε y_α y
                                         α_y^f ≡ λφzρy_ε y_α y: ρy_α (λψ:ψφz) y_ε y_α y
            z := y+3;                    A  ≡ λφzρy_ε y_α yx: φ y+3 ρy_ε y_α yx
                                         B  ≡ λφ: ε_y^f (Aφ)
            ...
         end block                       C, say
      end P;                             P  ≡ C(λρy_ε y_α y:ρI)

      begin integer u;
            P(u+x);                      D  ≡λφux: P(λψ:ψφu) ε_y^a ΩΩx
            ...                          ε_y^a ≡ λφuρy_ε y_α yx: ρI(λψ:ψφu) y_ε y_α u+x x
      end                                ...
end
```

In representing the procedure call in the above example, $(\lambda\psi{:}\psi\phi u)$ is passed as the return point argument instead of $\phi$, thus covering u. The application of the former to any LE uncovers u and restores the environment; e.g., in $\varepsilon_y^f$, the application is made to $y_\varepsilon$, and in P, to I. Note that in the representation of the procedure call, namely, D, we have used $\Omega$ for what would otherwise have been $\alpha_y^a$; this is so, because no assignment can be made to the particular actual parameter in this case.

The evaluation and assignment operators, both formal and actual, have been defined above slightly differently than in the two previous examples in which covering was not required. These two examples are worked out once again so as to make the treatment uniform whether or not covering is needed in a particular case.

Example

```
begin integer a;
   procedure P(x); integer x;
                                  ε^f_x ≡ λφρx_ε x_α x: ρx_ε (λψ:ψφ) x_ε x_α x
                                  α^f_x ≡ λφρx_ε x_α x: ρx_α (λψ:ψφ) x_ε x_α x
      x:= x+2                     A   ≡ λφρx_ε x_α xa: φρx_ε x_α x+2 a
                                  B   ≡ λφ: ε^f_x (A(α^f_x φ))
      ;                           P   ≡ B(λρx_ε x_α x: ρI)
      ...
      P(a);                       C   ≡ λφa: P(λψ:ψφ) ε^a_x α^a_x Ωa
   ...                            ε^a_x ≡ λφρx_ε x_α xa: ρI(λψ:ψφ) x_ε x_α aa
                                  α^a_x ≡ λφρx_ε x_α xa: ρI(λψ:ψφ) x_ε x_α xx
end
```

Example

```
begin integer u,v;
   integer procedure P(x,y); integer x,y; value y;
                                  ε^f_x ≡ λφρπx_ε x_α xy: ρx_ε(λψ:ψφ)πx_ε x_α xy
                                  α^f_x ≡ λφρπx_ε x_α xy: ρx_α(λψ:ψφ)πx_ε x_α xy
      begin
         P := x-y;                A ≡ λφρπx_ε x_α xyuv: φρ x-y x_ε x_α x y u v
                                  B ≡ λφ: ε^f_x(Aφ)
         x := y                   C ≡ λφρπx_ε x_α xyuv: φρπx_ε x_α yyuv
                                  D ≡ λφ: C(α^f_x φ)
      end compound                E ≡ λφ: B(Dφ)
   end P;                         P ≡ E(λρπx_ε x_α xy: ρIπ)
   ...
   u := P(v,u+1)+u;               F ≡ λφuv: P(λψ:ψφ)Ωε^a_x α^a_x Ω u+1 uv
                                  ε^a_x ≡ λφρπx_ε x_α xyuv:ρI(λψ:ψφ)πx_ε x_α vyuv
                                  α^a_x ≡ λφρπx_ε x_α xyuv:ρI(λψ:ψφ)πx_ε x_α xyux
                                  G ≡ λφπuv: φ π+u v
                                  H ≡ λφ: F(Gφ)
```

For subscripted variables occurring as actual parameters, the actual evaluation and assignment operators are again chosen so as to represent the fictitious assignments between the formal and actual parameter variables. But now this involves the LE's elem and replace introduced in the discussion of arrays (Sec. 2.12). We will simply illustrate the representation by means of an example.

Example

```
begin integer n;
   ...
   begin integer array x[1:n];
      procedure P(u,v); integer u,v;
         begin
         ...
            ...
         end P;
      begin integer y;
         ...
         P(y,x[y])
      end
   end
```

For the above program, the representation of the procedure calling statement P(y,x[y]) is the LE

$$\lambda\phi yxn\text{: } P(\lambda\psi\text{:}\psi\phi y)\varepsilon^{a}_{u}\alpha^{a}_{u}\Omega\ \varepsilon^{a}_{v}\alpha^{a}_{v}\Omega\ x\ n\ ,$$

where,

$$\varepsilon^{a}_{u} \equiv \lambda\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}vxn\text{: } \rho I(\lambda\psi\text{:}\psi\phi y)u_{\varepsilon}u_{\alpha}y\ v_{\varepsilon}v_{\alpha}v\ x\ n\ ,$$

$$\alpha^{a}_{u} \equiv \lambda\phi y\rho u_{\varepsilon}u_{\alpha}u\ v_{\varepsilon}v_{\alpha}v\ x\ n\text{: } \rho I(\lambda\psi\text{:}\psi\phi u)\ u_{\varepsilon}u_{\alpha}u\ v_{\varepsilon}v_{\alpha}v\ x\ n$$

$$\varepsilon^{a}_{v} \equiv \lambda\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}v\ x\ n\text{: } \rho I(\lambda\psi\text{:}\psi\phi y)\ u_{\varepsilon}u_{\alpha}u\ v_{\varepsilon}v_{\alpha}(x(\underline{\text{elem}}\ y\ n))\ x\ n\ ,$$

$$\alpha^{a}_{v} \equiv \lambda\phi y\rho u_{\varepsilon}u_{\alpha}uv_{\varepsilon}v_{\alpha}v\ x\ n\text{: } \rho I(\lambda\psi\text{:}\psi\phi y)u_{\varepsilon}u_{\alpha}u\ v_{\varepsilon}v_{\alpha}v(x(\underline{\text{replace}}\ y\ n\ v))n\ .$$

A procedure body may contain a procedure call, possibly a recursive one, in which the formal parameters are used in actual parameter expressions. And the parameters of the nested call may themselves be called by name. The representation in such a case requires covering of all the variables associated with the procedure body, including the local variables, the return variable, and the parameter variables. This is illustrated below.

Example

```
begin integer x;
   procedure P(y,n); integer y,n; value n;
      begin
         ...
      end P;
   procedure Q(z); integer z;
      begin integer w;
         P(z,x);
         ...
      end Q;
   ...
end
```

If the representation of the body of the procedure P is A, then the representation of P itself is

$$P \equiv A(\lambda\rho y_\varepsilon y_\alpha yn\colon \rho I) \ .$$

The representation of the statement P(z,x) is

$$\lambda\phi\colon \varepsilon^f_z(B\phi) \ ,$$

where $\varepsilon^f_z$ is the formal evaluation operator for z in Q, and B represents the call on P, as follows:

$$B \equiv \lambda\phi w\rho z_\varepsilon z_\alpha z\ x\colon P(\lambda\psi\colon\psi\phi w\rho z_\varepsilon z_\alpha z)\varepsilon^a_y\alpha^a_y\Omega x\ x \ ,$$

$$\varepsilon^a_y \equiv \lambda\phi w\rho z_\varepsilon z_\alpha z\ \rho_1 y_\varepsilon y_\alpha y\ x\colon \rho_1 I(\lambda\psi\colon\psi\phi w\rho z_\varepsilon z_\alpha z)y_\varepsilon y_\alpha z\ x \ ,$$

$$\alpha^a_y \equiv \lambda\phi w\rho z_\varepsilon z_\alpha z\ \rho_1 y_\varepsilon y_\alpha y\ x\colon \rho_1 I(\lambda\psi\colon\psi\phi w\rho z_\varepsilon z_\alpha y)y_\varepsilon y_\alpha y\ x \ .$$

The next example illustrates the representation of a procedure calling statement in which an actual parameter itself consists of a call on a procedure.

Example

```
begin integer x;
   procedure P(r,s); integer r,s; begin ... end P;

   procedure Q(t);   integer t;   begin ... end Q;

   begin integer y;
      ...
      P(x, Q(y));
   end
end
```

Because the second actual parameter, Q(y), in the above procedure calling statement P(x, Q(y)) does not require an assignment operator, the latter statement is represented by the LE

$$\lambda\phi yx\colon\ P(\lambda\phi\colon\psi\phi y)\varepsilon^{a}_{r}\alpha^{a}_{r}\Omega\ \varepsilon^{a}_{s}\Omega\ \Omega\ x\ .$$

The first actual parameter, x, poses no problem, other than the covering of the variable y not visible to the procedure declaration of P; hence, we define

$$\varepsilon^{a}_{r} \equiv \lambda\phi y\rho r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ x\colon\ \rho I(\lambda\psi\colon\psi\phi y)r_{\varepsilon}r_{\alpha}x\ s_{\varepsilon}s_{\alpha}s\ x\ ,$$

$$\alpha^{a}_{r} \equiv \lambda\phi y\rho r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ x\colon\ \rho I(\lambda\psi\colon\psi\phi y)r_{\varepsilon}r_{\alpha}r\ s_{\varepsilon}s_{\alpha}s\ r\ .$$

For the second actual parameter, Q(y), things are slightly more complex. (Note, however, that only an evaluation operator is needed in this case; the assignment operator is undefined.) First, we have to provide for a call on Q -- which requires covering all the variables associated with the call of P -- with the following actual evaluation and assignment operators:

$$\varepsilon^a_t \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s\ \rho_1\ \pi\ t_\varepsilon t_\alpha t\ x\colon\ \rho_1 I(\lambda\psi\colon\psi\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s)\pi t_\varepsilon t_\alpha y\ x$$

$$\alpha^a_t \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s\ \rho_1\ \pi\ t_\varepsilon t_\alpha t\ x\colon\ \rho_1 I(\lambda\psi\colon\psi\phi t\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s)\pi t_\varepsilon t_\alpha t\ x$$

Now, $\varepsilon^a_s$ is defined in terms of a call on Q, followed by an assignment of the resulting value to s, as follows:

$$A \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s\ x\colon\ Q(\lambda\psi\colon\psi\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s)\Omega\ \varepsilon^a_t \alpha^a_t \Omega\ x\ ,$$

$$B \equiv \lambda\phi y\rho r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha s\pi x\colon\ \rho I(\lambda\psi\colon\psi\phi y) r_\varepsilon r_\alpha r\ s_\varepsilon s_\alpha \pi\ x\ ,$$

$$\varepsilon^a_s \equiv \lambda\phi\colon\ A(B\phi)\ .$$

## 4.4 Label parameters

The representation of label parameters is actually much simpler than of the integer variety. The reason is that two different operations, evaluation and assignments, are possible with the latter type; in addition, the value of the parameter at any time has to also be carried along within the representation. In the case of a label parameter, the only possible actual operation is a jump to it. Thus, with each formal label parameter, p, we need to associate only one variable, usually denoted by $p_\gamma$, which is to be bound to the operator for effecting the actual goto operation. (The variable $p_\gamma$ is an element of the environment of the procedure in which p is declared.) Next, associated with each actual label parameter, and individual to each procedure call, is an LE that represents the parameter procedure to effect the jump to the actual label. For a parameter p, this "actual goto" operator is usually denoted by $\gamma^a_p$, with further distinguishing marks added when more than one procedure call is involved. Lastly, associated with each formal label parameter, and unique to each environment within the procedure body, is an LE, the "formal goto" operator, that represents a call on the actual parameter procedure, that is, an application of the actual goto operator; the formal

goto operator for the parameter p is denoted $\gamma_p^f$, again with further distinguishing marks added if more than one environment is involved.

For anyone who has followed the previous treatment of jumps (Sec. 3.3) and procedures with integer parameters (Sec. 4.3), the example below should suffice to explain how to represent label parameters.

Example

```
begin integer a;
   procedure R(v); label v;
      goto v;                    A ≡ γ_v^f ≡ λφρv_γ: ρv_γ(λψ:ψφ)v_γ
                                 R ≡ A(λρv_γ:ρI)
   begin integer b;
      procedure P(x,z); integer x; label z;
                                 ε_x^f ≡ λφρx_εx_αx z_γ:ρx_ε(λψ:ψφ)x_εx_αx z_γ
                                 α_x^f ≡ λφρx_εx_αx z_γ: ρx_α(λψ:ψφ)x_εx_αx z_γ
                                 γ_z^f ≡ λφρx_εx_αx z_γ: ρz_γ(λψ:ψφ)x_εx_αx z_γ
         R(z);                   B ≡ λφρx_εx_αxz_γba: R(λψ:ψφρx_εx_αxz_γb)γ_v^a a
                                 γ_v^a ≡ λφρx_εx_αxz_γbρ_1v_γa: γ_z^f φρx_εx_αxz_γb a
      begin integer c,d;
         P(d,L)                  C ≡ λφcdba: P(λψ:ψφcd)ε_x^a α_x^a Ωγ_z^a ba
                                 ε_x^a ≡ λφcdρx_εx_αxz_γba:
                                        ρI(λψ:ψφcd)x_εx_αdz_γba
                                 α_x^a ≡ λφcd ρ x_εx_αxz_γba:
                                        ρI(λψ:ψφcx)x_εx_αxz_γba
                                 γ_z^a ≡ λφcdρx_εx_αx z_γba: Lba
      end;
   L: ...
   end
end
```

## 5. CONCLUSION

By interpreting programming constructs intuitively as functions rather than machine commands, we have succeeded in modelling programming languages in pure lambda-calculus. We have been able to provide a nonimperative, completely descriptive representation of most of the important features of programming languages, including assignments, jumps, and procedures involving call-by-name and side-effects.

An immediate application of the model is in a simple semantic specification of programming languages in terms of lambda-calculus, achieving a standard of rigor that matches their syntactic specification. Of more interest, however, is the potential of this model in the study of properties of programs, in proving program equivalence and correctness, and in program simplification (source code level) and optimization (compiled machine code level). Since we describe a program as a lambda-expression, the above mentioned applications essentially reduce to transformations within lambda-calculus. The possibilities of some of these applications have been indicated in the body of the paper by means of examples.

## 6. REFERENCES


1. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," Comm. ACM 12, 10 (Oct. 1969), 576-580, 583.
2. Burstall, R. M., "Formal Description of Program Structure and Semantics in First-Order Logic," Machine Intelligence 5 (ed. Meltzer, B., Michie, D.) Edinburgh Univ. Press (1970), 79-98.
3. Manna, Z., Vuillemin, J., "Fixpoint Approach to the Theory of Computation," Comm. ACM 15, 7 (July 1972), 528-536.
4. Church, A., The Calculi of Lambda-Conversion, Princeton,1941.
5. Curry, H. B., Feys, R., Combinatory Logic, Vol. I, North-Holland, Amsterdam (1968).
6. Hindley, J. R., Lercher, B., Seldin, J. P., Introduction to Combinatory Logic, Cambridge Univ. Press, London (1972).
7. Landin, P. J., "A Correspondence between ALGOL 60 and Church's Lambda-Notation," Comm. ACM 8, 2-3 (Feb., March 1965), 89-101, 158-165.
8. Stratchey, C., "Towards a Formal Semantics," in Formal Language Description Languages (ed. Steel, T. B.), North-Holland, (1966), 198-220.
9. Orgass, R. J., Fitch, F. B., "A Theory of Programming Languages," Studium Generale 22 (1969), 113-136.
10. Reynolds, J. C., "Definitional Interpreters for Higher-Order Programming Languages," Proc. ACM Conf. (Aug. 1972), 717-740.
11. Abdali, S. K., "A Combinatory Logic Model of Programming Languages," in preparation.
12. Naur, P., et al., "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM 6, 1 (Jan. 1963), 1-17.
13. van Wijngaarden, A., et al., "Report on the Algorithmic Language ALGOL 68," Numerische Math. 14 (1969), 79-218.
14. Morris, J. H., "Lambda-Calculus Models of Programming Languages," Project MAC Report MAC-TR-57, MIT, Cambridge, Mass. (Dec. 1968).

15. Manna, Z., Ness, S., Vuillemin, J., "Inductive Methods for Proving Properties of Programs," SIGPLAN Notices 7, 1 (Jan. 1972) 27-50.
16. Cadiou, J. M., Manna, Z., "Recursive Definitions of Partial Functions and their Computations," SIGPLAN Notices 7, 1 (Jan. 1972), 58-65.
17. de Bakker, J. W., Recursive Procedures, Mathematical Centre, Amsterdam (1972).
18. Rosen, B. K., "Tree-Manipulating Systems and Church-Rosser Theorems," JACM 20, 1 (Jan. 1973), 160-187.
19. Kleene, S. C., Introduction to Metamathematics, van Nostrand, Princeton, New Jersey (1950).
20. Scott, D., "Outline of a Mathematical Theory of Computation," Proc. 4th Ann. Princeton Conf. on Info. Sci. and Sys. (1970), 169-176.
21. Scott, D., "Lattice Theory, Data Types, and Semantics," in Formal Semantics of Programming Languages, (ed. R. Rustin), Prentice-Hall, New Jersey (1972), 65-106.
22. Randell, B., Russell, L. J., ALGOL 60 Implementation, Academic Press, New York (1964).

PROF. E. ISAACSON
251 MERCER STREET